DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Programación SIMD

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Agustina Aldasoro	86/13	agusaldasoro@gmail.com
Maximiliano Rey	XXX/XX	mail
Ignacio Tirabasso	XXX/XX	mail

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Resumen

En el presente trabajo se describen los beneficios de la programación en Lenguaje Ensamblador bajo el modelo de programación SIMD mediante el uso de instrucciones SSE.

Índice

1. Objetivos generales

El objetivo de este Trabajo Práctico es evaluar la eficiencia del modelo de programación SIMD mediante la implementación de diversos algoritmos en lenguaje Ensamblador utilizando instrucciones SSE.

Las mediciones se realizan mediante pruebas empíricas del código frente a algoritmos que cumplen la misma especificación, implementados en un lenguaje de alto nivel (C).

En este proyecto, los algoritmos a implementar se basaron en el procesamiento de imágenes y video, en el cual el uso del modelo SIMD es provechoso.

2. Contexto

Abordando el objetivo de este trabajo, realizamos una experimentación enfocada en reducir el tiempo de cómputo de un programa basado en dos sucesos que tienen la capacidad de afectarlo.

Por un lado se encuentra la *capacidad de cómputo*, la cual limita la cantidad de operaciones aritméticas que el procesador puede paralelizar. Si el programa ejecuta un uso intensivo en operaciones aritméticas, al añadirle nuevas operaciones de esta índole se van a necesitar más ciclos de clock para ejecutarlas.

El otro cuello de botella importante es el *ancho de banda de la memoria*. Cuando el programa ejecuta una instrucción que implica un acceso a memoria, se precisan más ciclos de clock para que la memoria responda, en particular si el dato no se encuentra en el cache.

Diseñamos nuestros casos de testeo con el fin de observar para cada programa cuál es el factor que determina el tiempo de cómputo.

Nuestra experimentación se centra en la cantidad de ciclos de clock que transcurren desde el inicio hasta el final de la ejecución del programa. Se adjunta con la documentación los archivos *.py utilizados para calcular la esperanza y la varianza de cada una de las mediciones.

3. Enunciado y solucion

3.1. Mediciones

Realizar una medición de performance *rigurosa* es más difícil de lo que parece. En este experimento deberá realizar distintas mediciones de performance para verificar que sean buenas mediciones.

En un sistema “ideal” el proceso medido corre solo, sin ninguna interferencia de agentes externos. Sin embargo, una PC no es un sistema ideal. Nuestro proceso corre junto con decenas de otros, tanto de usuarios como del sistema operativo que compiten por el uso de la CPU. Esto implica que al realizar mediciones aparezcan “ruidos” o “interferencias” que distorsionen los resultados.

El primer paso para tener una idea de si la medición es buena o no, es tomar varias muestras. Es decir, repetir la misma medición varias veces. Luego de eso, es conveniente descartar los outliers [1], que son los valores que más se alejan del promedio. Con los valores de las mediciones resultantes se puede calcular el promedio y también la varianza, que es algo similar el promedio de las distancias al promedio[2].

Las fórmulas para calcular el promedio μ y la varianza σ^2 son

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$

[1] en español, valor atípico: `http://es.wikipedia.org/wiki/Valor_atpico`

[2] en realidad, elevadas al cuadrado en vez de tomar el módulo

3.2. Filtro *cropflip*

Programar el filtro *cropflip* en lenguaje C y luego en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 1.1 - análisis el código generado

En este experimento vamos a utilizar la herramienta `objdump` para verificar como el compilador de C deja ensamblado el código C.

Ejecutar

```
objdump -Mintel -D cropflip_c.o
```

¿Cómo es el código generado? Indicar *a*) Por qué cree que hay otras funciones además de `cropflip_c` *b*) Cómo se manipulan las variables locales *c*) Si le parece que ese código generado podría optimizarse

Experimento 1.2 - optimizaciones del compilador

Compile el código de C con flags de optimización. Por ejemplo, pasando el flag `-O1`[3]. Indicar 1. Qué optimizaciones observa que realizó el compilador 2. Qué otros flags de optimización brinda el compilador 3. Los nombres de tres optimizaciones que realizan los compiladores.

Experimento 1.3 - calidad de las mediciones

1. Medir el tiempo de ejecución de cropflip 10 veces. Calcular el promedio y la varianza. Consideraremos outliers a los 2 mayores tiempos de ejecución de la medicion y también a los 2 menores, por lo que los descartaremos. Recalcular el promedio y la varianza después de hacer este descarte. Realizar un gráfico que presente estos dos últimos items.

   Luego de ejecutar 10 veces el filtro Cropflip obtuvimos los siguientes resultados:

ASM	**C**
70.925	1.152.187
70.521	1.151.544
32.859	761.937
43.720	649.248
64.236	1.152.847
70.793	1.153.061
71.271	1.152.765
44.616	1.152.798
56.124	725.420
71.775	1.155.718
Esperanza	
59.684	1.020.752,5
Desvío Standar	
13.720,5329	203.626,443

   El cuadro denota la cantidad de ciclos de clock utilizada por cada ejecución del programa.

   Luego de eliminar los dos valores más altos y los dos valores más bajos, recalculamos obteniendo los siguientes datos:
   Esperanza: 37.721,5 (ASM) y 652.407,8(C)
   Desvío Standar: 31.706,23 (ASM) y 544485,4142(C)
   Se puede ver que al eliminar los outliers, la esperanza y la Desvío Standar tienen valores más cercanos.

[3] agregando este flag a `CCFLAGS64` en el makefile

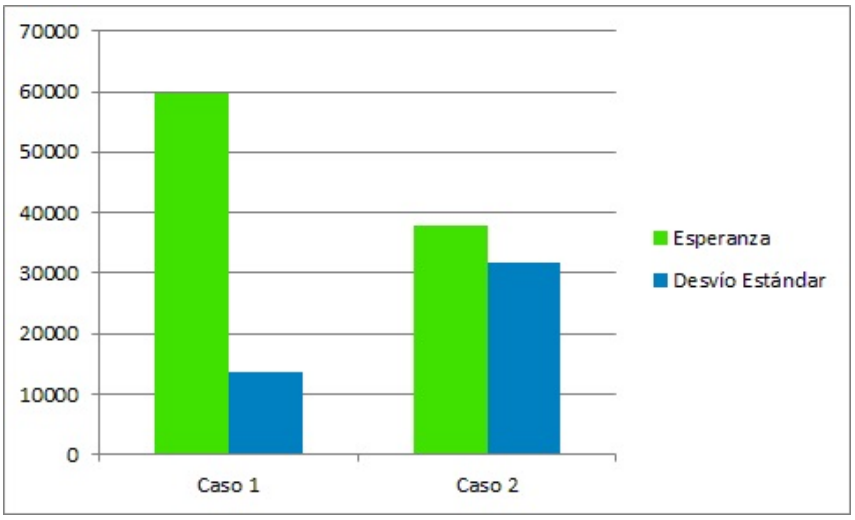

Figura 1: Assembler

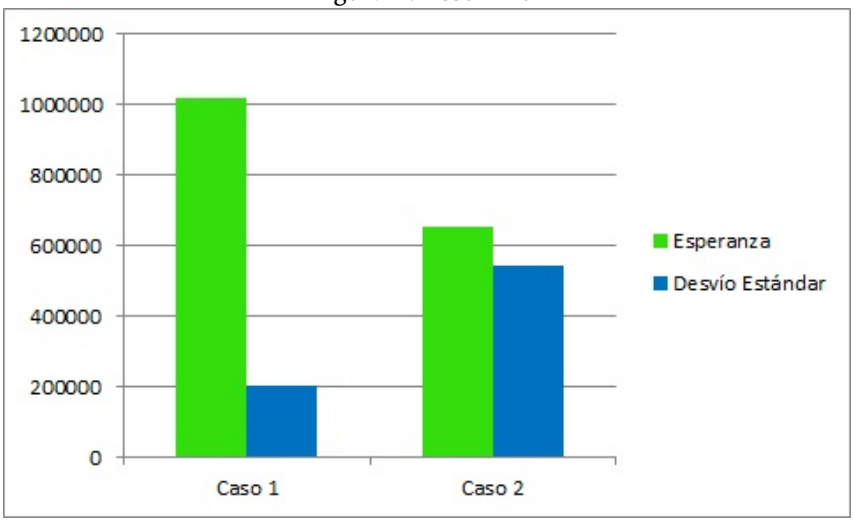

Figura 2: C

Siendo el Caso 1 las mediciones de esperanza y Desvío Standar para todos los casos de test y el Caso 2 las mediciones sin tener en cuenta los cuatro outliers.

2. Implementar un programa en C que no haga más que ciclar infinitamente sumando 1 a una variable. Lanzar este programa tantas veces como *cores lógicos* tenga su procesador. Medir otras 10 veces mientras estos programas corren de fondo. Realizar los mismos casos de experimentación que en el ejercicio anterior.

Los resultados obtenidos en esta experimentación fueron menores que los anteriores:

ASM	**C**
33.585	542.928
33.798	544.155
33.402	544.857
33.228	543.687
33.159	543.252
33.441	543.324
34.089	544.224
33.768	760.359
34.563	542.448
34.473	542.982
Esperanza	
33.750,6	565.221,6
Desvío Standar	
465,49	65.049,34

Luego de eliminar los dos valores más altos y los dos valores más bajos, recalculamos obteniendo los siguientes datos:
Esperanza: 20.208,3 (ASM) y 326.162,4(C)
Desvío Standar: 16.501,015 (ASM) y 266.310,725(C)
Acá también se puede apreciar que al eliminar los outliers, la esperanza y la Desvío Standar tienen valores más cercanos.

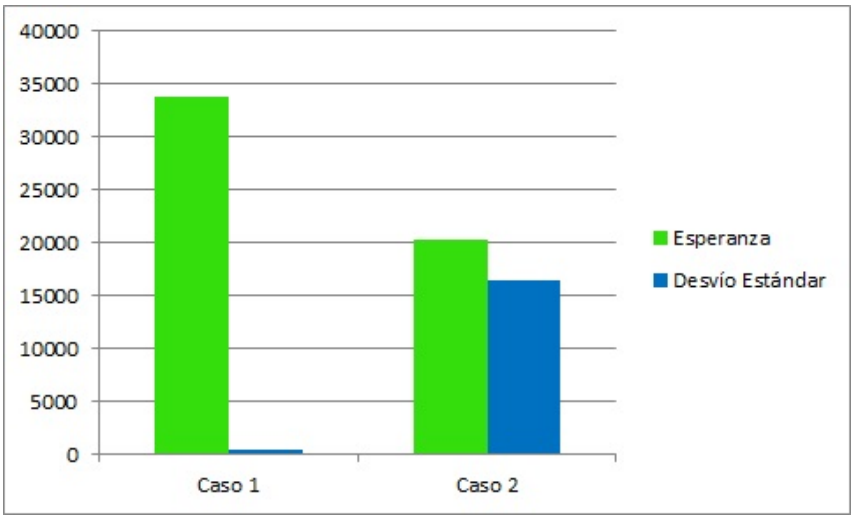

Figura 3: Assembler

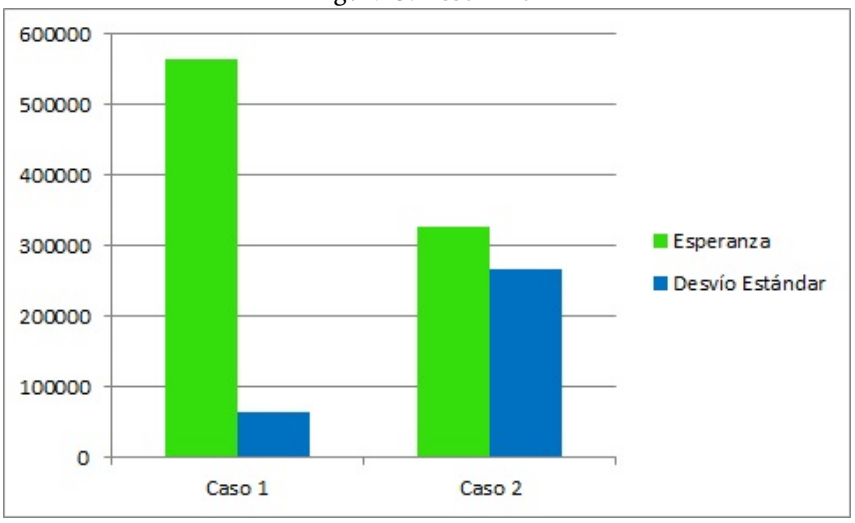

Figura 4: C

Siendo el Caso 1 las mediciones de esperanza y Desvío Standar para todos los casos de test y el Caso 2 las mediciones sin tener en cuenta los cuatro outliers. Se puede observar que las mediciones mejoran con la ejecución del ciclo infinito de fondo, esto se debe a que .... ??????????????

A partir de aquí todos los experimentos de mediciones deberán hacerse igual que en el presente ejercicio: tomando 10 mediciones, luego descartando outliers y finalmente calculando promedio y Desvío Standar.

Experimento 1.4 - secuencial vs. vectorial

En este experimento deberá realizar una medición de las diferencias de performance entre las versiones de C y ASM (el primero con -O0, -O1, -O2 y -O3) y graficar los resultados.

Experimento 1.5 - cpu vs. bus de memoria

Se desea conocer cual es el mayor limitante a la performance de este filtro en su versión ASM.

¿Cuál es el factor que limita la performance en este caso? En caso de que el limitante fuera la intensidad de cómputo, entonces podrían agregarse instrucciones que realicen accesos a memoria extra y la performance casi no debería sufrir. La inversa puede aplicarse, si el limitante es la cantidad de accesos a memoria. [4]

[4]también podría pasar que estén más bien balanceados y que agregar cualquier tipo de instrucción afecte sensiblemente la performance

Realizar un experimento, agregando 4, 8 y 16 instrucciones aritméticas (por ej `add rax, rbx`) analizando como varía el tiempo de ejecución. Hacer lo mismo ahora con instrucciones de acceso a memoria, haciendo mitad lecturas y mitad escrituras (por ejemplo, agregando dos `mov rax, [rsp]` y dos `mov [rsp+8], rax`).[5]

Realizar un único gráfico que compare: 1. La versión original 2. Las versiones con más instrucciones aritméticas 3. Las versiones com más accesos a memoria

Acompañar al gráfico con una tabla que indique los valores graficados.

3.3. Filtro *Sierpinski*

Programar el filtro *Sierpinski* en lenguaje C y en en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 2.1 - secuencial vs. vectorial

Analizar cuales son las diferencias de performace entre las versiones de C y ASM de este filtro, de igual modo que para el experimento 1.4.

Experimento 2.1 - cpu vs. bus de memoria

¿Cuál es el factor que limita la performance en este filtro? Repetir el experimento 1.5 para este filtro.

3.4. Filtro *Bandas*

Programar el filtro *Bandas* en lenguaje C y en en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 3.1 - saltos condicionales

Se desea conocer que tanto impactan los saltos condicionales en el código de filtro Bandas con `-O1` (la versión en C).
Para poder medir esto de manera aproximada, remover el código que detecta a que banda pertenece cada pixel, dejando sólo una banda. Por más que la imagen resultante no sea correcta, será posible tomar una medida aproximada del impacto de los saltos condicionales. Analizar como varía la performance.

Experimento 3.2 - secuencial vs. vectorial

Repetir el experimento 1.4 para este filtro.

3.5. Filtro *Motion Blur*

Programar el filtro *mblur* en lenguaje C y en ASM haciendo uso de las instrucciones **SSE**.

Experimento 4.1

Repetir el experimento 1.4 para este filtro

4. Conclusiones y trabajo futuro

[5]Notar que en el caso de acceder a `[rbp]` o `[rsp+8]` probablemente haya siempre hits en la cache, por lo que la medición no será de buena calidad. Si se le ocurre la manera, realizar accesos a otras direcciones alternativas.